1975

A process controller for a hierarchical process structured operating system

Kral, Theodore Carl

Monterey, California. Naval Postgraduate School

A PROCESS CONTROLLER FOR A HIERARCHICAL PROCESS STRUCTURED OPERATING SYSTEM

Theodore Carl Kral

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A PROCESS CONTROLLER FOR A HIERARCHICAL PROCESS STRUCTURED OPERATING SYSTEM

by

Theodore Carl Kral

June 1975

Thesis Advisor: B. E. Allen

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A PROCESS CONTROLLER FOR A HIERARCHICAL PROCESS STRUCTURED OPERATING SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Master Thesis; June 1975
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Theodore Carl Kral		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1975
		13. NUMBER OF PAGES 83
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Operating System; Processor Management

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of this thesis is to report the design and implementation of a process controller which would allow the concurrent execution of real-time, timesharing, and batch processes. The design was to be compatable with the UNIX multiprogramming timeshared system and run on the PDP 11/50 computer. A summary of the documentation of UNIX, which was a prerequisite to design, is presented as a prelude to the design. The design and criteria for choosing it are also presented, followed by the implementation and related problems.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
(Page 1) S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (con't)
Finally, conclusions drawn and suggested improvements of the final product are discussed.

A Process Controller for A Hierarchical Process Structured Operating System

by

Theodore Carl Kral
Lieutenant, United States Navy
B.S., United States Naval Academy, 1969

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1975

ABSTRACT

The purpose of this thesis is to report the design and implementation of a process controller which would allow the concurrent execution of real-time, timesharing, and batch processes. The design was to be compatable with the UNIX multiprogramming timeshared system and run on the PDP 11/50 computer. A summary of the documentation of UNIX, which was a prerequisite to design, is presented as a prelude to the design. The design and criteria for choosing it are also presented, followed by the implementation and related problems. Finally, conclusions drawn and suggested improvements of the final product are discussed.

TABLE OF CONTENTS

LIST OF DRAWINGS

I. INTRODUCTION

The Computer Science Department of the Naval Postgraduate School acquired in the fall of 1974 two PDP 11/50 computers and various associated peripherals. The intent was to integrate this new computer hardware with the present laboratory hardware to produce a computing system which could handle a wide variety of functions from normal batch operation to real-time and graphics. However, due to limited funds and the unavailability of a suitable off the shelf machine compatable operating system, only a minimum amount of software was acquired with the majority of funds available being used for hardware acquistion. The operating system chosen was UNIX, a monoprocessor multiprogramming timesharing operating system, developed by Bell Laboratories [1]. It was realized from the start that this operating system probabily would not, as acquired, satisfy the needs of the proposed system. As a result of this a Computer Science Department group project was undertaken to transform the basic UNIX operating system into one which more fully met the requirements of the proposed system. This paper in conjunction with Refs. 2 - 4, presents some of the preliminary work undertaken to implement multiprocessing [2], a virtual machine monitor [3], a hierarchical memory manager [4], and a new processor manager.

One goal of the proposed computer system was the capability to concurrently handle real-time, timesharing, and batch processes. The intent of this paper is to report the work performed in the design and implementation of a process controller to satisfy this goal.

For the sake of clarity, this paper begins by presenting some general concepts and views on what makes up a process controller. For the most part it is a condensation of the views and ideas presented in Refs. 5 - 7. Thus, if a more complete or detailed description or elaboration is desired in any area, the reader's attention is referred to these references, in particular Ref. 5. The first chapter presents the concepts of batch, multiprogramming, timesharing, and real-time as pertaining to process control in an attempt to establish a common ground with the reader in these areas. The remainder of the chapter consists of an examination of several currently operational operating systems and their respective methods of process control. The first, MULTICS, is presented due to its notariety and reputation in the field of timesharing and multiprogramming. The other, EXEC 8, is presented due to its implementation of the goal in mind, i.e. handling of real-time, timesharing, and batch processes concurrently. The systems presented are only representative examples of the implementation of a process controller and are by no means the only ones in the field [8,9].

Since the basic role of an operating system is to act as an interface between the user and the computer hardware, any work on an operating system requires at least a basic insight into the computer's operation. Thus, Chapter III, presents an overview of the PDP 11/50 hardware. In addition, process management is only one piece of the whole operating system and must interface smoothly with the other portions if the whole is to have any chance of success. Therefore, an understanding of the UNIX operating system and in particular the functioning of the present process controller was required before attempting any revision. This, however, proved to be a formidable task. The lack of sufficient documentation degenerated this effort in many instances to interpretation of raw code just to gain a

general understanding of the system's operation. The results of this effort are presented in Chapter IV with emphasis being placed on process control.

Chapter V presents the prosposed design and considerations taken into account. Chapter VI presents the actual implementation of the design and some of the major problems encountered in this area. Chapter VII gives an analysis of the status of the system with the new process controller implemented. It also presents conclusions drawn and an insight to some possible further improvements.

II. PROCESS MANAGEMENT

A. GENERAL

There are many ways to conceptually view an operating system. However, a recent view presented by Madnick and Donovan in Ref. 5 is one of the better and is the method adopted by this paper. Essentially, a functional approach is taken viewing an operating system as the management of physical resources: memory, processors (CPU's), information (programs and data), and I/O devices (disks, tapes, teletypes, etc.). All the requirements of an operating system are considered to belong in one of these four management catagories. The basic unit to which all the managers respond is called a process. A process (or task) is a computation that may be done concurrently with other computations. Each manager must keep track of the status of each resource, decide which process is to get the resource (how much and when), allocate it, and eventually reclaim it.

There is not normally a clean division of responsibilities and functions among the managers. Due to this and the fact that there is constant interaction between managers as a process passes through its various stages of execution, an operating system is further characterised as a hierarchical structure. Figure 1 presents such a structure commonly called a ring structure [5]. This is one way of conceptualizing the hierarchical concept, but a tree structure [9] or something similiar could also be used. The

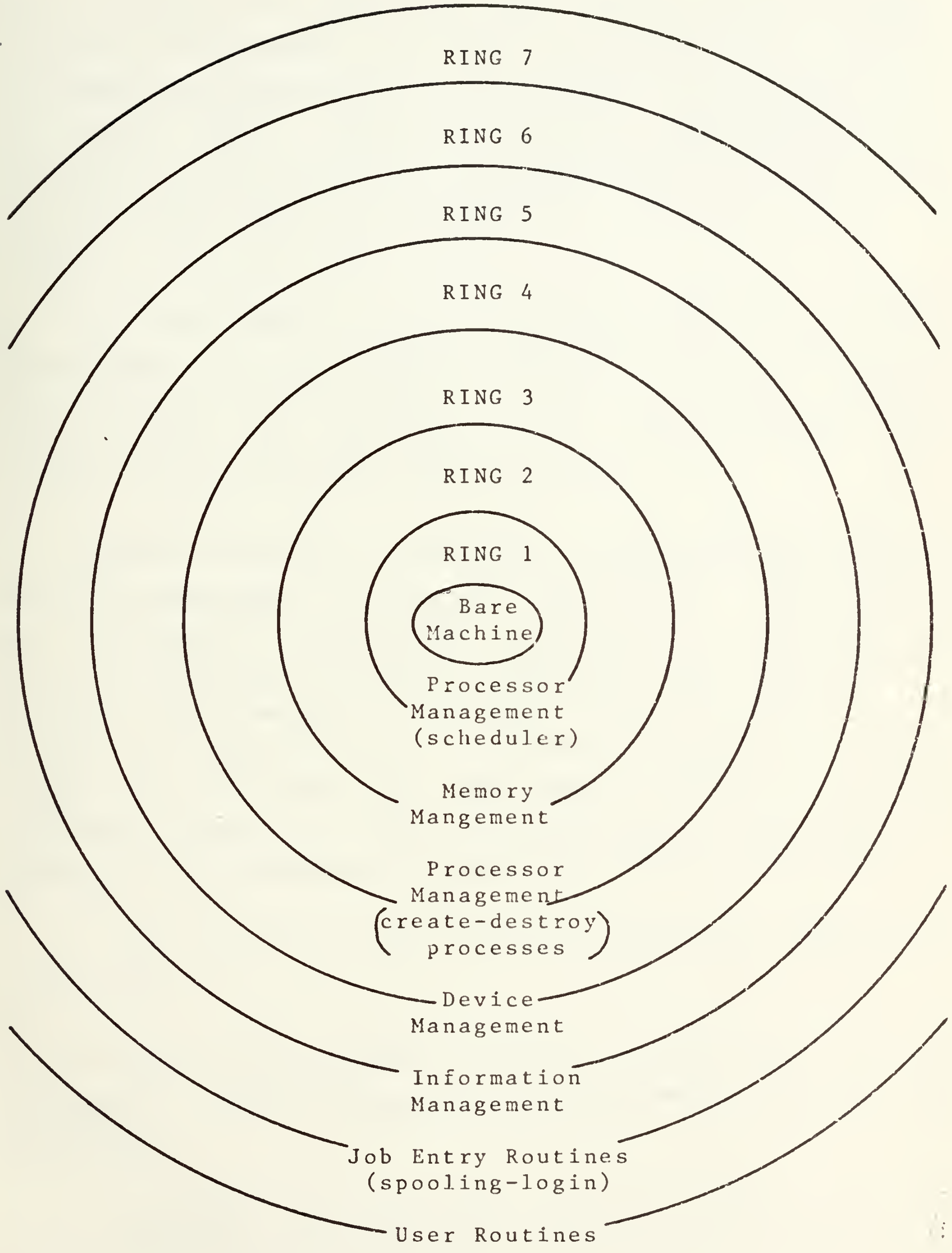

Fig. 1 Ring Structured Operating System

main function of overlaying this concept onto the manager concept is to restrict the flow of control. That is, members of rings can request service from members of inner rings but not outer rings. Essentially, outer ring members are permitted to give orders to any one who is on the same ring or on an inner ring but are permitted only to respond to requests from outer ring members.

Since this paper concerns itself primarily with process management, a more detailed understanding of the basic functioning of the processor manager is needed. Further insight into the operation of the other managers can be gained by the interested reader from Ref. 5.

Process control or processor management is concerned with the management of physical processors, specifically, the assignment of processors to processes. In conjunction with this function, processor management must monitor and control all processes and jobs within the system (a job being the collection of activities needed to do work required by a user and can consist of more than one process). There are four main functions which must be performed to accomplish the processor manager's task [5]:

* Keep track of the processors and the status of processes.

* Decide which process will have a chance to use the processor, which process gets the processor, when, and for how long.

* Allocate the processor to a process by setting up necessary hardware registers.

* Reclaim the processor when a process relinquishes processor usage, terminates, or exceeds allowed amount of

usage.

Processor management can be further subdivided into two modules: a job scheduler which creates the processes and a process scheduler which decides which process receives a processor, at what time, and for how long.

1. Job Scheduler

The job scheduler can be considered a macroscheduler concerned with performing the following tasks [5]:

* Keeping track of the status of all jobs. That is, which jobs are attempting to get service and the status of jobs being serviced.

* Choosing the policy by which jobs will be allowed entry into the system.

* Allocating the necessary resources for the scheduled job by calls to the memory manager, device manager, file manager, and process scheduler.

* Deallocating these resources when the job is done.

Essentially the job scheduler chooses some subset of jobs submitted, lets them into the system, creates processes from them, and assigns these processes some resources.

2. Process Scheduler

The process scheduler is a microscheduler concerned with performing the following tasks [5]:

* Keeping track of the state of the processes. That is, is process running on CPU, performing I/O, loaded in memory, etc.

* Deciding which process gets a processor, when, and for how long.

* Allocating processors to processes.

* Deallocating processors from processes.

The process scheduler allocates processors among the subset of processes allowed into the system by the job scheduler.

B. MULTIPROGRAMMING

This section will not attempt to fully explain the concept of multiprogramming; the interested reader is referred to Refs. 5 and 6. The main purpose here is to note that the process scheduler only takes on significant value when considering processor management for a multiprogramming system. The characteristic of a multiprogramming system is the concurrent core residence and interleaving execution of two or more programs [6]. In a monoprogramming system there is only one process allowed into the system at a time by the job scheduler. Therefore, the function of the process scheduler for all practical purposes is minimal and normally is considered a part of the job scheduler's function. However, the implementation of multiprogramming forces the job scheduler to allow more than one process into the system at once. This creates the requirement to manage two different groups of active items concurrently. The addition of this complexity on the process level makes the division

of processor management into the separate areas of job scheduler and process scheduler benefical.

C. EFFECT OF JOB STREAM

At present, processor managers are designed to handle one or more of the following basic job classes: batch, timesharing (interactive), and real-time. The names of the classes are not as important as the characteristics they represent. The prominent features unique to each class are the ones which the processor manager must allow for or make use of, as the case may be. Therefore, since the processor manager's basic function is to control (manage) jobs and processes, these characteristics are quite important. It is the intent of this section to point out some of these features for each broad job catagory and how they affect processor management.

1. Batch

Batch jobs inherit the name of the first type of operating system, the serial batch system [7]. In this type system the user entered his job to the oprating system and was not involved with the job again until job execution was completed.

The concept is carried over for this classification, in that batch jobs and related processes are normally self contained (operate independent of user). This type of job, once entered into the system, requires no other contact with the user until completion, and only then if some output is generated. In addition, these type jobs are normally time independent in that the system can delay execution or

interrupt their execution for long periods of time without affecting the job's results. Essentially timing is not a critical factor in program execution, and delays are acceptable. These jobs are flexible in their system requirements, and the system can manipulate their execution to suit overall performance.

2. Timesharing

This classification of jobs became popular with the advent of the timesharing system. A better name for this type might be interactive, and both names will be used interchangeably throughout this paper. The concept of timesharing is a subject within itself, and there are numerous articles, books, etc. on the subject throughout the literature. Therefore, this paper will not attempt to fully cover the concept, but as an aid to understanding this job classification, a very brief overview of the timesharing operation is presented.

When timesharing, multiple users concurrently engage in a series of interactions with the system via on-line terminal devices. The user is able to respond immediately to system queries, and similarly, the system can respond immediately to requests for service by the user. Actually, the timesharing control program is designed to service many users on a multiplexed basis; each user being given a time slice of available CPU time. One user's think and reaction times are used to perform work for other users. Thus, a user has the operational advantage of seeming to have a machine to himself, and a trade off is made between efficient CPU utilization and effective user service. This system is user oriented with jobs and processes being created in an ad hoc manner.

Timeshared jobs and related processes are characterized by the interaction between the process and the user during execution, usually through the use of a terminal device. A timeshared process normally requires series of short bursts of computations and then goes blocked for I/O to await new commands from the user. Due to the fact that the computer is reacting to a human input directly, time plays an important role in the execution of this type job. This process can not be ignored by the system for extended periods of time, as with batch, because the user will become impatient. However, the wait time he will endure is large when compared with the operating speed of the computer, and execution of this process can be delayed slightly. Essentially this process works on a "hurry up and wait" principle. The user gives his command; the process must hurry to execute this command, hurry to give an answer back to the user, and then wait for a longer period than it took to execute. The user expects the computer to respond immediately to his request, and then wait patiently until he is ready to give another.

3. Real-time

There are numerous definitions of real-time, but regardless of which one is chosen, time is the crucial governing factor. Thus, efficient and rapid scheduling takes on an important meaning for this type of process. Long wait times for executing real-time processes are unacceptable. If a real-time process is not run as soon as ready, it probably will not execute properly. In many cases this type of job or process can be considered a glorified timesharing process for it is highly responsive to input/output as is timesharing, but differs in that the initiator of the input and the receiver of the output are not necessarily human. In addition, there is a greater time

constraint on the operating system, in particular the process controller.

D. IMPLEMENTATION

This section concerns itself with the implementation of processor management. Its intention is to give the reader a general feel for some methods employed in the implementation of processor management. Some specific examples of processor management as implemented on presently operational operating systems are also presented. The choice of MULTICS is due to its notariety and EXEC 8 due to its concurrent handling of real-time, timesharing, and batch jobs. References 7 and 8 contain discussions of other systems if the reader is interested.

1. General

There are numerous methods, schemes, or algortihms which can be used to implement a process controller. One method for keeping track of jobs is to create a data structure called a job control block (JCB). The JCB can contain various information but usually has some provision for the status and position of the job in the job queue. Another method is to maintain lists or tables for the different possible states, status, etc. of jobs in the system. A third method is to implement a combination of these two methods.

Selection of a policy for entering jobs into the system is open ended and can be as simple or complex as desired. One policy used on simple systems is to have an operator select which jobs will enter the system based on

some criteria (friends first, number of resources requested, preset priorities, system balance, etc.). Another policy is to have the operating system perform the selection based on some similar criteria. A third might be a combination of these two. Another criteria commonly employed is first-in-first-out (FIFO) with an upper limit on the number of jobs allowed entry into the system or allow jobs to enter until memory is full.

Similar structures as those used for maintaining the status of a job can be used for processes. The physical allocation of the processor to the process is governed by hardware characteristics as well as individual tastes. Reference 5 lists some of the typical process scheduling policies employed to decide which process gets the processor as:

(1) Round Robin. Each process in turn is run to a time quantum limit. DEC PDP/8 Timesharing System (TSS/8) utilizes this method [7].

(2) Inverse of Remainder of Quantum. A process is placed back on the ready list in a position relative to the amount of quantum time left. If half is used, it is placed in the middle. If all is used, it goes to the end.

(3) Priority. The highest priority ready job is selected.

(4) Limited Round Robin. Processes are run round robin for a fixed number of times. Then they are run only if there are no other ready processes in the system.

(5) Preferred Treatment to Interactive Jobs. Processes which are actively conversing with the user

(interactive processes) are given preference. These processes are run immediately after user input in order to provide the initial quick response expected by the user.

(6) Merits of Job. In this method the system itself assigns priorities based on certain characteristics of a process. One example would be to assign high priorities to processes performing short jobs. Another would be to give preference to processes performing I/O.

In addition, there are many different combinations and derivatives of the above methods in use today. This is due to attempts to capitalize on the effect of different job types, etc. Thus, a priority method might be used to select processes for running, and a round robin method might then be utilized in case of ties.

2. MULTICS

The Multiplexed Information and Computing System (MULTICS) [5,10] was the result of a cooperative effort of the Massachusetts Institue of Technology, Bell Laboratories, and the computer department of the General Electric Company (now part of Honeywell Information Systems Incorporated). It is a large scale, timeshared, general-purpose information system utility run on the HIS 6180. MULTICS permits concurrent batch and timesharing operations, and it incorporates many unique concepts along with other concepts that had previously seen only limited use. An overview of its process control method is presented; the interested reader is referred to Ref. 10 for more complete information on this system.

MULTICS processor management is composed of modules that handle the transition between process states as shown

in the state diagram in figure 2.

Good response time is insured by limiting the number of users allowed entry into the hold state as indicated by the number one. All users are given weights according to certain merits (job, class of user, etc.) measured in units. The operator of the system sets the maximum number of units allowed on the system depending on the system's configuration at the time. A certain number of units is allotted to groups of users called load control groups. Each MULTICS user belongs to one of the groups and is either privileged or nonprivileged within the group. Privileged users may preempt nonprivileged users. If a user is on the system and the group's units are exceeded, he is automatically logged out after a grace period.

The number two represents the operation of the job scheduler which transfers jobs from the hold to the ready state. The hold state consists of a set of n queues, one per priority level. Interactive processes are initially assigned to one of a range of high priority levels while batch jobs are assigned to one of a range of lower priority levels. The higher priority levels within the batch range overlap into the lower levels of the interactive range. The highest priority level is given to processes that are expected to make brief use of the processor (short interactions). Lower priority levels are assigned to longer running interactive processes and for short batch or background jobs. Long batch jobs would be assigned to the lowest levels. The highest priority process moves from the hold state to the ready state when its working set can fit into memory. A noneligible process may preempt an eligible process in order to give good response to interactive users issuing commands of short duration. The preempted process is rescheduled by being placed on top of the priority level queue from which it came with a time allotment equal to what

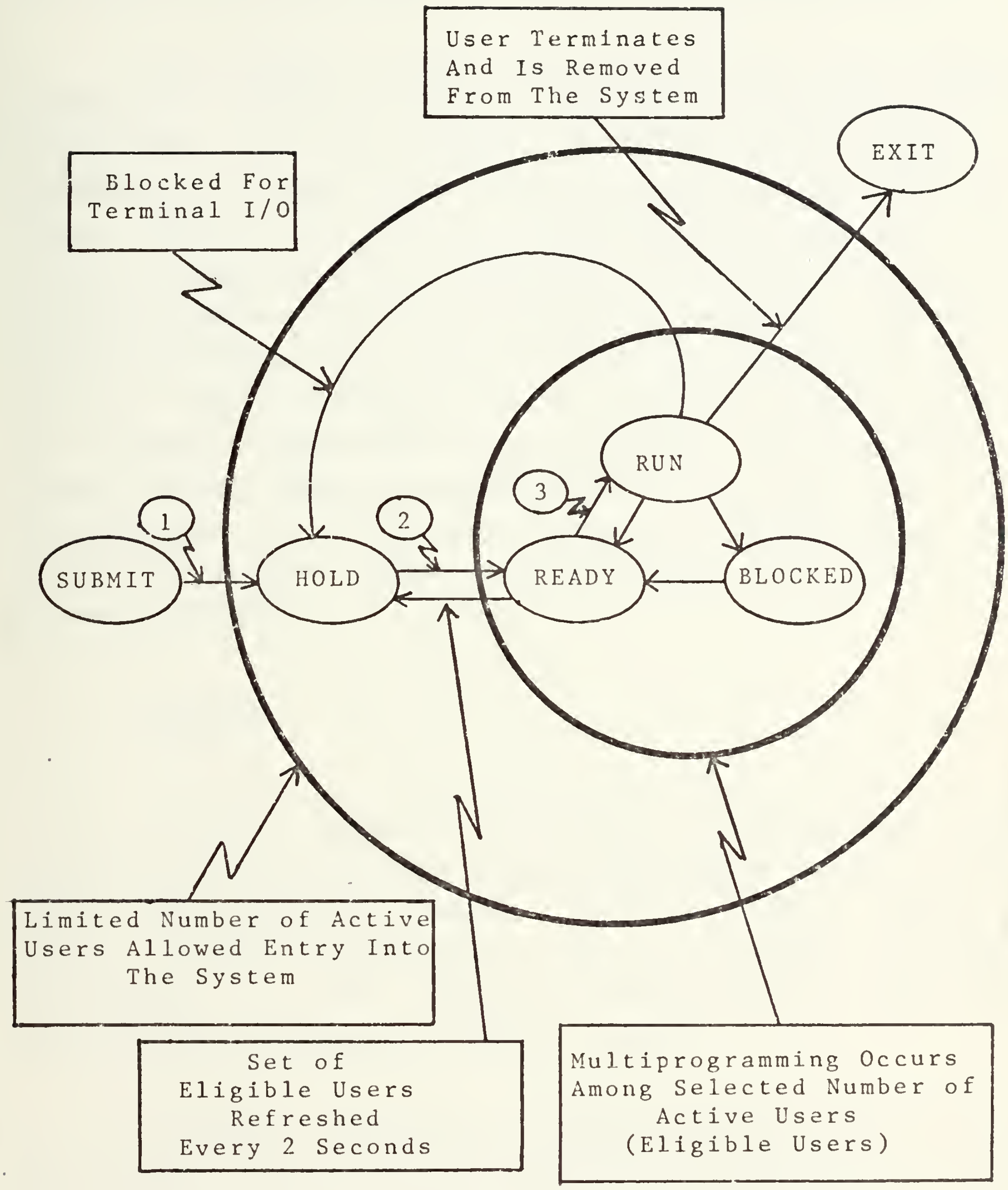
User Terminates And Is Removed From The System
EXIT
Blocked For Terminal I/O
RUN
3
1
2
SUBMIT
HOLD
READY
BLOCKED
Limited Number of Active Users Allowed Entry Into The System
Set of Eligible Users Refreshed Every 2 Seconds
Multiprogramming Occurs Among Selected Number of Active Users (Eligible Users)

Fig. 2 State Diagram of MULTICS

ever time is still unused from its last time quantum.

The number three corresponds to the process scheduler. MULTICS assigns priorities to processes in the ready queue, and the process scheduler always gives the processor to the ready process with the highest priority. When a process enters a wait state for the occurrance of a system event, the CPU will be given to the ready and eligible process that has the highest priority. When the awaited event occurs, the waiting process is readied, and if it has a higher priority than the currently running process, the running process yields the processor. It is possible at times for all eligible processes to be in the wait state simultaneously. If this case occurs, the processor is given to an idle process which is always ready and always has the lowest priority.

3. EXEC 8

EXEC 8 operates as a master control program which establishes the multiprogramming environment for the UNIVAC 1108 multiprocessor system. EXEC 8 permits concurrent batch, demand (interactive), and real-time processing operations. A more complete description of the entire system may be gained from Refs. 7 and 8.

Seperate scheduling philosophies are used for each job-process mode. For batch processing, all submitted jobs are collected in groups arranged by priorities. All jobs within a high priority group must be initiated before jobs from lower priority groups will be selected. The order of initiation within each group is based first on the resources available at the moment and second on the order of job submission. Exceptions to this policy are batch jobs which must be completed before a specified time of day (called a

deadline). Jobs with an impending deadline may be selected at any time regardless of priority. The estimated running time (supplied by the programmers) of the active jobs in the system is used to determine when a deadline job should be initiated. If a deadline cannot be met via normal scheduling, the system will take the necessary action to insure the required completion time if possible. This action in some instances may cause a degradation in the normal multiprogramming operation as well as increasing system overhead.

Demand processing jobs are initiated immediately upon arrival into the system. Demand jobs are normally scheduled among themselves on a round robin basis. A priority is also assigned to each demand job but is only used when overload situations exist to give certain privileges in CPU assignment.

Real-time programs are submitted for processing in a manner similiar to batch job processing. However, once initiated, real-time programs receive an execution priority directly below the interrupt processing priority. They are also locked in core even though they may be initially in a dormant state awaiting receipt of selected external interrupts.

Executing programs may also specify the initiation of a subtask to be executed concurrently with the main task. The subtask to be executed may be initiated immediately or delayed for a given increment of time depending on the request.

For each job initiated, the scheduler prepares a program control table. This table contains information such as run ID, estimated run time, the current facilities assigned, and the core requirements of the particular task

being executed. The control table is maintained by the dynamic allocator during the execution of the task and returned to the scheduler when the task terminates.

Due to the high priority allotted real-time programs, they are allocated CPU time whenever they are ready to use it. Since several programs with real-time requirements may have identical priorities, these programs must share control as required. If a program is interrupted due to an I/O completion for a higher priority program, the CPU will be given to the high priority program.

EXEC 8 maintains two dispatching queues, one for batch job tasks and one for demand job tasks. When a batch job task is ready for execution, it is placed in a core queue and executed in the general mix of batch tasks. For demand jobs a task is introduced onto a special queue and is given core space and CPU time as soon as its turn comes up. The basic philosophy of the scheduler is to meet required deadlines for batch jobs, while at the same time maintaining the required response time for demand users. Within this dynamic operating environment the dividing line between demand and batch programs is subject to constant change as emphasis is placed on batch runs approaching required completion time.

The job scheduler prepares a switching list used by the process scheduler in switching control among core resident programs as various events and contingencies arise. The allocator periodically adjusts the switching level of programs or classes of programs so as to force the CPU time to be used in a particular manner based on deadlines, priorities, and interaction rates, as well as certain overall constraints as to how CPU time should be shared among the different types of programs. The overall constraints (the portion of time to be spent between demand

and batch processing) are specified at system generation or via operator direction.

III. OVERVIEW OF THE LABORATORY HARDWARE

At the time of writing, not all of the proposed hardware had been received or installed into the computer laboratory; figure 3 shows the proposed laboratory equipment and configuration. However, during design and implementation of the new process controller the operational equipment consisted of two PDP 11/50 CPU's, 3 magnetic tape units, 2 DEC writers, 3 RK disk drives, 32K of CSPI memory, 32K MOS memory, 96K core memory, 1 line printer, and various types of on-line terminals. The majority of peripherals were attached to one CPU; with the second CPU having access to 1 RK disk drive, 1 DEC writer, 1 paper tape reader/punch, and 32K of core memory. A brief description of the UNIBUS, PDP 11/50 computer, and memory units utilized follows. A more comprehensive description of this equipment as well as the other peripheral equipment in figure 3 can be found in Refs. 11 and 12.

A. UNIBUS

The PDP 11/50 utilizes a bidirectional, asynchronous bus, called a UNIBUS, to communicate with its many devices. The UNIBUS is the key to the PDP 11's architecture. All system components are connected to it and utilize it to communicate with each other in identical fashion. Devices are connected through hardware registers to the UNIBUS. Any device (except memory) can dynamically request the UNIBUS to transfer information to another using a scheme based on real

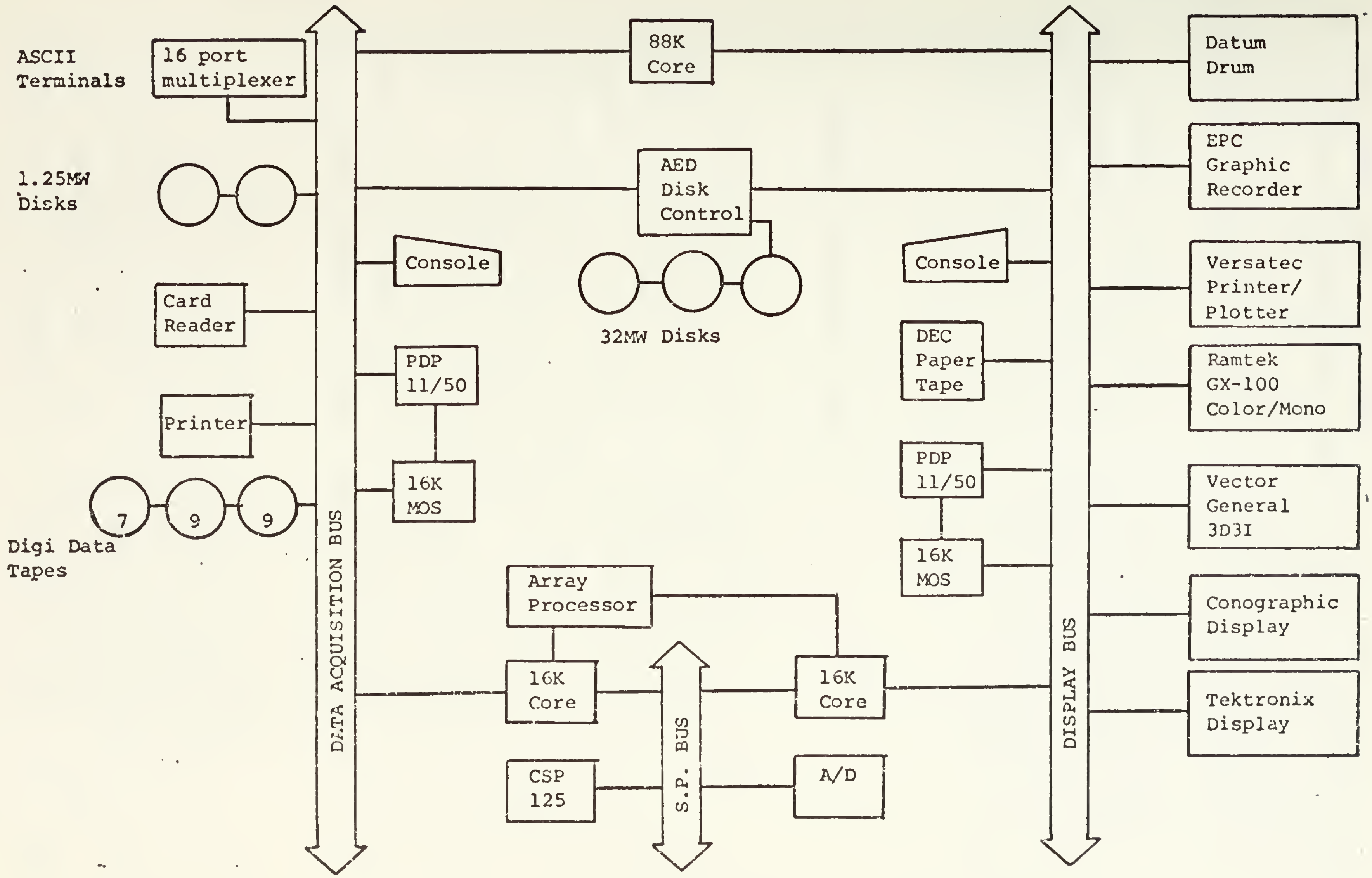

Fig. 3 PDP Laboratory Hardware Configuration

and simulated core locations. All of the device registers are located within the address space. Thus, peripherals appear to the CPU just as memory locations with special properities. This allows a common set of instructions to be used for operations both on memory and peripherals.

Devices communicate on the UNIBUS in a master slave relationship. Duing bus operation, one device has control of the bus. This device, called the master, controls the bus when communicating with another device called the slave. A priority structure determines which device has control of the bus at any given instant of time.

B. CENTRAL PROCESSING UNIT

The PDP 11/50 is a medium scale general purpose 16 bit computer manufactured by Digital Equipment Corporation. Besides performing all the arithmetic and logical operations required in the system, the central processor acts as an arbitration unit for UNIBUS control by regulating bus requests and enforcing the priority control structure. The machine operates in three modes: Kernel, Supervisor, and User. When in Kernel mode, a program has complete control over the machine; when the machine is in any other mode the processor is inhibited from executing certain instructions and can be denied direct access to peripherals on the system.

The central processor contains 16 general registers which can be used as accumulators, index registers, or as stack pointers. One of the general registers, R7, is used as the program counter. Three others are used as Processor Stack pointers, one for each operational mode. The remaining 12 registers are divided into two sets of

unrestricted registers, R0-R5. The current register set in operation is determined by the Processor Status Word.

The Processor Status Word (PS) contains information on the current status of the CPU. This information includes the register set currently in use; current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

The PDP 11/50 implements a priority interrupt feature. The central processor operates at any of eight levels of priority, 0-7. When the CPU is operating at level 7, an external device cnnnot interrupt it with a request for service. The current priority is maintained in the PS. A special instruction is provided by the system to allow dynamic alteration of the priority level.

Since system intercommunication is carried out through the UNIBUS, a device must gain control of the UNIBUS to interrupt program execution and force the processor to branch to a service routine. There are two sources of interrupts, hardware and software. A hardware interrupt occurs when a device wishes to indicate to the program or central processor, that a condition has occurred (such as data transfer complete, end of tape, etc.). The interrupt can occur on any one of the four Bus Request levels. A software interrupt is generated by the programmer setting a bit in the high order byte of address location 777772 octal.

Interrupt handling on this machine is automatic; no device polling is required to determine which service routine to execute. Interrupts are serviced as follows [11]:

* Processor relinquishes control of the bus, priorities permitting.

* When a master gains control, it sends the processor an interrupt command and a unique memory address which contains the address of the device's service routine in Kernel virtual address space, called the interrupt vector address. Immediately following this pointer address is a word (located at vector address+2) which is to be used as a new Processor Status Word.

* The processor stores the current Processor Status Word (PS) and the current Program Counter (PC) into CPU temporary registers.

* The new PC and PS (interrupt vector) are taken from the specified address. The old PS and PC are then pushed onto the current stack as indicated by bits 15, 14 of the new PS and the previous mode in effect is stored in bits 13, 12 of the new PS. The service routine is then initiated.

* The device service routine can cause the processor to resume the interrupted process by executing the return from interrupt instruction, which pops the two top words from the current processor stack and uses them to load the PC and PS registers.

C. MEMORY UNITS

The present system has three different types of memory which can be connected in various configurations. These are MOS with a cycle time of 495 nsec, Computer Signal Processing Incorporated memory (CSPI) with a cycle time of 750 nsec, and dual-ported core with a cycle time of 900

nsec. The central processor communicates directly with the MOS through a very high speed data path which is internal to the CPU. The remainder of memory is accessed via the UNIBUS.

In addition to the three different speeds of memory, there is also a memory management unit. This unit contributes to the processors ability to operate in three different modes: Kernel, Supervisor, and User mode. The memory unit maintains three separate sets of 32 sixteen bit registers; one for each mode of operation. Each set is divided into two groups of 16 registers. One group is used for all instruction fetches, index words, absolute addresses, and immediate operands. The other group is used for all other references. Each of these groups is further subdivided into two sections of 8 registers. One section is the Page Address Registers which are used to convert virtual addresses to physical addresses. The other section is the Page Descriptor Registers which contain information relative to page expansion, page length, and access control.

Operation of the memory management unit causes an address to be interpreted as a virtual address instead of a direct physical address. This virtual address is used to construct a new 18 bit physical address as follows: The high order three bits of the sixteen bit address word are used to determine which Page Address Register is to be used, and the other thirteen bits determine the amount of displacement required within the page. Implementation of this device expands the maximum memory space addressable from 64K bytes to 256K bytes.

IV. OPERATING SYSTEM UTILIZED - UNIX

The operating system acquired to run the PDP 11/50 hardware was UNIX. As acquired, UNIX contained very little documentation. As a result, considerable time and effort was expended studying the C language [13] and in several cases assembly language [14] code just to gain a general understanding of the system operation. This general understanding of the whole system was necessary in order to isolate those portions pertinent to process control and determine their interaction with other portions. Further study of these individual modules was then required to gain the detailed understanding required before a design could be conceived.

This chapter is concerned with providing a general understanding of the UNIX operating system as delivered before any changes were implemented. Detailed discussion is restricted to those areas of the system related to process control with emphasis placed on areas directly concerned with changes made. However, some information on the whole operating system is presented as background. The interested reader is directed to Refs. 1 - 4 for additional information.

A. GENERAL

UNIX is a general purpose, multiuser, interactive operating system designed for use on the Digital Equipment

Corporation PDP-11/40, 11/45, and 11/50 computers and is a product of Bell Laboratories. The greater part of UNIX is written in the C language [13,15]. The remainder of UNIX is written in assembly language [14] which until the summer of 1973 was the only language used. It was during this period that the system was rewritten in C to make it much easier to understand and modify, besides including many functional improvements, such as multiprogramming and the ability to share reentrant code among several user programs. The resident portion of the operating system occupies 42K bytes of memory. The system was primarily designed for interactive use with its most important role, as viewed by its creators, being to provide a hierarchical file system incorporating demountable volumes [1]. In addition it offers several other features, including (1) compatable file, device, and inter-process I/O; (2) the ability to initiate asynchronous processes; (3) system command language selectable on a per-user basis; and (4) over 100 subsystems including a dozen languages.

B. PROCESS CONTROL

1. Keeping Track of Processes

a. Process Make Up

Under UNIX a process consists of a computer execution environment, referred to as an image, and a corresponding process control block. The image is the current state of a pseudo computer and includes a core image, general register values, status of open files, current file system directory, etc. The user core part of

an image is divided into three logical segments. The program text (code of program) begins at location 0 in the virtual address space, and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates. The remainder of the information (general register values, status of open files, etc.) is contained in a 1K block unique to each process, called the u-vector. The first portion of this block is a C language structure called "user". The remainder of this block is used for a stack. This stack is utilized by the system as a Kernel stack when the process is running or the system is executing in its behalf. The prominent feature of this block is that all the information needed to run a process can be retrieved from this 1K storage area. This facilitates the assignment and removal of processes from the CPU. It, in conjunction with the process control block, is the system's means of controlling and maintaining a process from creation to termination.

In addition to the image, the system also maintains a process control block for each process. This again is a C language structure called "proc" (see Appendix A). However, unlike the user structure which is attached to the executeable code in user's memory (but not the user address space), the process control block is fixed in the system's section of memory. Addressing of these blocks is absolute and while their contents may change, their true location is stationary throughout the running of the system. UNIX provides for fifty process control block structures. This physically limits the number of active processes to a maximum of fifty with two of these, process block 0 and

process block 1, consumed by the system for special purposes (will be discussed later). Process control blocks are used by the system to maintain information on the status of the process (loaded in core, ready, blocked, etc.), its priority, user ID, corresponding image address (core or disk), etc. Essentially, it contains information on the general status of the process. The system relies totally on the information contained in this structure to determine which process will be allocated the CPU in a multiprogramming environment.

b. Creation of a Process

Except while UNIX is bootstrapping itself into operation, a new process can come into existence only by the use of a "fork" system call (see Appendix C). When the "fork" is executed by a process, the process is split into two independent executeable processes. The two processes have independent copies of the original core image, and share any open files. A separate process control block is allotted to the new process. The newly created process is referred to as the child, and the old process is referred to as the parent. A link is maintained from the child to its parent via the parent's process ID number ("p_pid") which is stored in the child's process control block as the value of "p_ppid". If the parent should terminate before the child, the child's link is changed to the ID number of process control block one, INIT. Maintenance of this parent/child structure enables the system to provide additional features for the user, in particular the system call, "wait" (see Appendix C). This feature is also the basis for the system's method of interaction with the user as presented in a later section.

c. Process Entry

UNIX communicates with the user through terminals which are the primary means of directly entering user processes into the system. There are two main routines utilized by the system to assist in this direct communication and process entry. One is called INIT, and the other is called SHELL. There also exists a third method to interact with the system, but by a less direct means. This means is afforded the user through the use of system calls within his program code. The operation of these three types of interfaces follows.

(1) INIT. Upon initial execution INIT creates one process (through a "fork" system call) for each active terminal or channel. These child processes are used to handle the login sequence and open the appropriate typewriters for input and output. Each of these processes then types out a message requesting login and waits, reading the typewriter. When a user types his name and hits the carriage return, the corresponding version of INIT is awakened, reads the name, checks it against a user list, requests a password (if required), and checks the password's validity. If the login procedure is correct, this child of INIT changes its process user ID to that of its user, makes other changes related to a user in the u-vector, and performs an "exec" of the SHELL. The "exec" is a system call presented in Appendix C. It essentially causes the present process to be replaced by the process passed as an argument. In this case, the child of INIT is transformed into the process called SHELL (explained below). If a child of INIT terminates through user typing an incorrect login or the SHELL terminating, the parent INIT process is awakened and recreates a new child process for that terminal. Reference 1 provides additional information on INIT.

(2) SHELL. The SHELL is an interactive user process designed to respond to a specific set of interactive commands. Reference 16 contains a complete listing and description of the commands available. The SHELL is a child of INIT and is unique to the terminal assigned it by INIT. Upon execution, the SHELL types the prompt character ("%"), performs a read on the input buffer, and waits for input. Upon receiving a command, it checks it for validity against various lists, and then creates a child process to execute the process corresponding to the command. The SHELL process, as parent, then waits for the child to finish before looping back to the beginning of its code to type the prompt and perform the read again. That is, it waits unless the user specifies for it not to wait. In that case, it loops back immediately upon return from the "fork" call. The user indicates for it not to wait by typing an "&" immediately after the command. This has the effect of executing the command in a background mode divorced from the actions of the terminal. For more information on the SHELL, the reader is directed to Ref. 1.

(3) System Calls. UNIX provides the user with a number of system calls which can be used to communicate indirectly with the system. Several calls, particularly those affecting the process control, with discriptions are contained in Appendix C. A complete list is contained in Ref. 17. System calls enable the user to make use of system routines to perform such normally restricted functions as create another independent process during program execution, completely change the code to execute during program execution, perform input/output operations, etc. Execution of a system call causes an interrupt which turns control over to the operating system for performance of the desired action. At completion of the request, control is returned to the user process.

2. Scheduling of Processes

a. Classification

Although UNIX provides for the creation of background and foreground processes, it makes no distinction between the two types once they are entered into the system. That is, there is no set label, identification, or priority specifically assigned to the process with which the system could classify the process. Instead, UNIX utilizes the process's execution to indirectly perform this classification. This is done by assigning a default priority to all processes entering the system and constantly changing and resetting this priority throughout the life of the process's execution. This is done in a manner which gives preference to interactive and short executing processes.

b. Scheduling

Processes are allowed entry into UNIX through the use of the "fork" system call as long as there are process control blocks available, and there is room in memory or on the swap device to hold the process. Upon creation, a process will be assigned core if memory space is available and assigned to the swap device if memory is not available.

The actual scheduling of processes is accomplished under UNIX through the use of two schedulers. One is a high level scheduler which bases selection on a FIFO method. The other can be considered a low level

scheduler which bases selection strictly on priority.

The high level scheduler determines who will be loaded into core and who will be left on the swap device. This scheduler essentially regulates the set or list of processes which the lower level scheduler can consider for running. A process is selected for placement in memory from the swap device based on the amount of time it has been waiting. The process which has been waiting the longest is the first to be loaded into memory (FIFO) when enough memory becomes available. Once a process is swapped into memory, it does not become a candidate for removal until a certain quantum (2 sec) is exceeded. It is not actually swapped out unless a process has been waiting for memory in excess of a minimum wait time (3 sec). Processes which are in the SWAIT state, waiting for a resource, are candidates for removal regardless of time. However, processes which are locked in memory for I/O, etc. are not candidates for removal until unlocked. A flag is set ("runnin") to indicate that a process is on the swap file and for some reason could not be entered into memory. At least every one sixtieth of a second (sometimes sooner), this flag is checked. If it is set, the high level scheduler is called. The high level scheduler then applies its algorithms to update the lower level scheduler's list of candidates for running. Upon completion of its tasks, the high level scheduler passes control to the lower level scheduler for selection of the next process to run.

The low level scheduler searches the entire list of processor control blocks, looking for the highest priority process and ignoring those process which are not in a run state or not loaded in memory. A process is considered not in the run state if it is blocked for I/O, waiting on another process, or in some similiar dormant state. The low level scheduler starts its search with the

process block which is next in numerical sequence after the process block belonging to the process previously selected. It concludes the search with the process block belonging to the process previously selected. A decision is made between equal priority processes by selecting the first process encountered in the search. This can be considered a round robin method overlayed on process priority; for barring any changes in priority levels, the next time a selection is to be made, the search will start after the process which was just selected. Therefore, that process will not be selected again until every other process of equal priority has been selected.

These two schedulers are not employed by UNIX with any regularity. Instead, asynchronous occurrances such as creation of a process, clock pulse when processor running in user mode, device becoming available, etc. are used to initiate these routines. Essentially, any occurrance which might cause a change in the state of some process or the system will cycle the system through its scheduling routines.

The UNIX routines SCHED and SWTCH perform the jobs of high level and low level scheduler respectively. Both of these routines are explained in Appendix B, and the reader is encouraged to read these discriptions for a better understanding of the handling of process selection under UNIX.

c. Setting Priorities

As mentioned, UNIX essentially utilizes the priority method for determining which process to assign to the CPU next. Therefore, since priorities play such an important role in the execution of a process, it is of prime

importance to understand how UNIX assigns these priorities.

Priorities are based on a whole number scale between -127 to +127 with lower values representing higher priorities. At creation, a process is given a default priority (100). From that point on this value is changed numerous times depending on the execution of the process. At every clock interrupt (every 1/60 sec) the process running on the CPU at the time has its priority decreased (value incremented) by one if it is in the user mode (CPU not executing in Kernel mode), and its present priority is higher than a preset minimum (value less than 105). Thus, if left alone, eventually all priorities would end up at a value of 105. However, UNIX avoids this by constantly resetting priorities when certain actions are performed by the process during execution. If a process is required to wait on a device for some reason, it is given a very high priority. A process waiting on an I/O buffer for instance is given a priority of -50. Thus, when the buffer is ready, this process will probabily be immediately chosen for running, enabling it to read or write the buffer as need be. However, upon doing this, the process is required to envoke a system call or cause some other interrupt which causes the UNIX routine, TRAP, to be called. One of the final duties performed by the TRAP routine after it has satisfied the cause for the interrupt is to reset the priority of the process presently assigned to the processor back to its default value (100). There are other priorities used to reflect other process states as well as reasons for entering TRAP. Therefore, the reader is encouraged to read the description of the operation of the UNIX routines, SLEEP, WAKEUP, and TRAP, presented in Appendix B for more information in this area. The result of this constant incrementing and resetting of the priority has the effect of causing noninteractive processes on the whole to have lower priorities (greater than 100) than interactive processes.

It also insures immediate service to those processes, forced to wait on a device, upon that device becoming available. Of course, a noninteractive process which makes numerous system calls will maintain a certain high priority status, but this will only last for one clock pulse and at best, be equal to the priority of the interactive processes. Thus, under UNIX, assignment of the CPU is favored toward active interactive processes and becomes less favorable as a process does more and more computatations.

3. Assignment of CPU

Actual assignment of a process to the CPU is done by a UNIX routine called SWTCH (see Appendix B). This routine utilizes two other routines; SAVU to actually remove the old process and RETU to initiate the new process. Both routines are described in Appendix B. This procedure of assigning the processor to a process is facilitated greatly by the u-vector block described previously. The address of the beginning of this block which is the beginning of the "user" structure is constantly maintained in the process control block as "p_addr". When a process is to be removed from the CPU, all its registers, stack pointer, and other important data are stored in this u-vector which, as noted, remains a part of the process code. Thus, when a process is assigned to the CPU, the "p_addr" in the process block is used to locate this block, and all information needed to run the process is retrieved from this block.

4. Termination of a Process

Upon termination of a process, the system makes a copy of the u-vector out onto the swap device, sets the address of this location into the "p_addr" of the associated

process control block, changes the status to an inactive state, and frees all other resources maintained by this process. The system then locates the parent of this process, and if a "wait" system call was used by the parent, the system transfers certain information to the parent's u-vector, such as execution times, child's ID, child's return argument, etc. The system then frees the child's process control block and swap space utilized for the u-vector. At this point, the terminated process no longer exists except for the information stored in the parent's u-vector. If a "wait" was not performed by the parent, no copy of this information is retained, and all resources including the process control block are released immediately.

C. NORMAL PROCESS EXECUTION

The process's creation begins with a program source file which has normally been created through use of the SHELL and EDIT commands [16]. The source file is then compiled by the corresponding language compiler, and an object file is created. The user then commands the SHELL process to execute this file, and the birth of a process begins. The SHELL does a "fork". UNIX complying with the "fork" call sets up a separate process control block, user block, and copy of code. This copy of the image is either produced in core or if no room exists the copy is made on the swap file. The child process, which is exactly the same code as the SHELL, then attempts to perform the system call "exec" with the new process as an argument. UNIX complies with the "exec" and replaces the SHELL's child's code with that of the passed argument making updates in the process control block and image as necessary. Allowances for differences in code size, etc. are also taken care of by the system. This

new process which is still a child of the SHELL, is then placed in the ready state (assuming it was created in memory). Meanwhile, the SHELL performs a "wait". The newly created process can now become active, and will probabily be picked to run after the SHELL performs the "wait". At the next clock pulse, the process's execution may be halted temporarily while the low level scheduler selects the highest priority process from those available to run. If its priority is less than or equal to another ready process, the processor is given to the other process. The sample process is then placed back on the ready list. At the next clock pulse or sooner, if conditions warrant it, the sample process will be swapped out of core, removed from the ready state, and placed in the not loaded condition. Eventually, the process will be swapped back into core and given the CPU again. If the sample process does some I/O and goes blocked, its priority is reset to 100. Thus, at completion of the I/O, the sample is near the top of the list for receiving the CPU. However, if the device was busy, the sample process will be placed in a special state which causes it to receive a very high priority . However, while in this state it is not eligible to run on the processor. When the device becomes available, the sample process is placed back in the ready state along with all the other processes waiting for this device. They all also have the same high priority, and therefore, the CPU is given to one of these for running. The one selected immediately goes blocked for I/O, and the CPU is systematically passed to each of the other processes which causes them to be placed back into the special state until the device is available again. Eventually the sample process will be selected, complete its I/O, and terminate. At termination, UNIX notifies the SHELL who has been patiently waiting for its child to finish. The system then frees memory, and the process control block of the sample process. The SHELL then completes the cycle by returning to the user.

V. DESIGN

A. GOAL AND SUBGOALS

The basic goal was to design and implement a process controller which would enable the concurrent execution of batch, timesharing, and real-time processes within the UNIX operating system. In conjunction with attaining the major goal certain other subgoals were established.

1. Preservation of Unix

It was realized that operating systems are complex and portions are not designed, implemented, and debugged overnight but take considerable time. UNIX presently had a tested and functioning process controller which handled an interactive environment containing many desireable features. Therefore, if the new process controller could be designed around the techniques presently employed by UNIX, the final interfacing with the system would be less painful and time consuming. Since time was a governing factor, this approach was adhered to as much as possible.

2. Memory Conservation

The amount of free storage space available for addressing in the Kernel mode of operation was limited with

UNIX presently consuming a large portion of it. In addition, other modifications either were or would be made to the system, so conservation of space was a significant factor.

3. Make Design Simple and Direct

The new process controller was just a preliminary implementation, and changes would have to be made as requirements and performance standards became clearer. Therefore, future modifications needed could be more easily recognized if the present design was simple and direct. This was not a significant goal, carrying less importance than others, but did contribute in the final design.

4. Integration With Other Modifications

Concurrent with the implementing of the new process controller was a conversion of the system to multiprocessing, implementation of a hierarchical memory manager, and implementation of a virtual machine to name a few of the many changes being made to UNIX. Thus, compensation for any modifications or needs induced by these designs had to be absorbed in the design of the processor controller. The converse, of course, was also true. Therefore, intercommunication was of prime importance, especially in the area of memory management and multiprocessing.

B. PROPOSED DESIGN

As mentioned previously, UNIX arrived with very little

documentation. As a result only a sparse amount of knowledge was available on the operating system's design at the time of the basic goal's conception. Therefore, as a prerequisite to design and implementation of the goal, a comprehensive study of the system was required, especially in the area of process control. This was mainly to determine what UNIX already provided and what modifications would be necessary, if any. A summary of the results of that effort are presented in Chapter IV. After having acquired a more complete knowledge of UNIX, the following design was proposed.

1. General

The present handling of interactive and noninteractive (batch) processes by UNIX would be preserved. Since, at present, the only means of entering jobs into the system was via terminals, it was determined that the present system's handling of this feature was adaquate and in some cases desirable. It was realized that this was not truly a separate classification of batch processes at present, but the background mode could be used to achieve the same desired CPU utilization as was intended by purely batch processing. Through the use of the system call "nice" [17], which lowers the default priority of a process, a batch (background) process can be forced to run in a lower priority range than interactive processes. Thus, UNIX provides a means of separating out a batch class through priorities. A lower range of priorities can be assigned to the desired classes of batch jobs. This range could even be allowed to extend into the interactive process's present priority range of 100 to 105. The lower priority would distinguish the batch processes as a separate class to be run only if no interactive processes are ready to run. The creation of a separate INIT/SHELL combination with the card

reader as input file and the printer as the output file would be one method of implementing the entry of of this new class into the system. However, at present no firm commitments were desired to be made as to a method or means of alternate entry for batch jobs or processes. Thus, the present scheduling and operation of UNIX for these two classes of processes would be altered only to the extent necessary to handle the requirements imposed by a real-time process.

2. Scheduling

The two levels of scheduling utilized by UNIX would also be used. The operation of the high level scheduler would be unchanged except for the fact that real-time processes would be locked in memory and never be candidates for swapping out. This locking would be contingent on memory manager's implementation.

The low level scheduling policy, the one employed to actually assign the CPU to a process would be modified as follows: CPU would be assigned to a real-time process whenever a real-time process was ready to run. If more than one real-time process was ready then the one with the highest priority would receive the processor. If priorities were equal then real-time processes would share processors on a round robin basis. If no real-time process was ready to run then processors would be assigned to highest priority ready process. Setting of priorities of all processes including real-time would be the same as that presently implemented by UNIX.

3. Keeping Track of Processes

A similar method of keeping track of processes as that employed by UNIX would be used by the new process controller with the exception that an additional entry would be added to the process control block called "p_rtflag". This flag would be set to indicate real-time status of the corresponding process.

Entry of the processes into the system would employ the same technique as presently employed by UNIX. Once inside the system and upon execution, a process would request real-time status via a system call, "rtime". This routine would ensure all provisions for real-time status were met. If conditions were unsatisfactory (see Real-time Restrictions section), the routine would not grant the request and would return to the process a value indicating reason for not granting the real-time request. A return of zero will be given if the bit was set and the process was made real-time. Until this system call is properly executed, real-time processes would be treated the same as all other processes under UNIX.

4. Assignment of CPU

No changes were deemed necessary in the procedures utilized by UNIX for the physical assignment of the processor.

5. Termination of Process

Routines involved with termination of processes under UNIX (EXIT and WAIT) would be modified to ensure unsetting of real-time bit in the process control block. Outside of this, all processes would terminate as under UNIX. A routine called, "nonrtime", would be provided as a

system call to enable a user to voluntarily cancel real-time status if no longer required.

6. <u>Real-time Restricitions</u>

Due to the highly demanding requirements which real-time processes place on a system, certain features were designed into the system primarily to add additional protection to the system from malfunctioning programs.

* A variable would be used to define the number of real-time processes allowed to be active concurrently in the system. This number could be easily modified as conditions required, and initially it would be set to a value of one. Enforcement of this policy would be accomplished within the RTIME routine. This limitation was required due to the fact that too many real-time processes running at once would eliminate any service to other processes in addition to degrading the real-time service. Also, since real-time processes were to be locked in memory, the danger existed that too many real-time processes executing at once would use up all or a significant portion of memory. This condition would prevent the effective swapping of processes into and out of memory which in turn would lead to decreased CPU utilization, especially if all the real-time processes went blocked for similar reasons.

* Real-time status would be automatically lost if system call, "exec" is utilized, but status could be regained through new request upon execution of new process. This is to prevent an unsavory user from attempting to by-pass checks established in RTIME routine by utilizing status of previous real-time process. Appropiate code would be introduced in the routine corresponding to this system call to turn the real-time bit off.

* All child processes produced by real-time processes would be non real-time, but would be permitted to request real-time status. Similar reasons as for the first two restrictions required this one. Appropiate code would be introduced into the routine corresponding to the "fork" system call.

* Preemption on I/O devices was not provided. Real-time processes would be given head of the line privileges with respect to devices, but would not be permitted to take devices away from other processes except as provided for by UNIX. It was felt that the majority of time critical devices utilized by real-time processes would be dedicated. Those that weren't, probobly would not be utilized by a non-real-time process, if utilization of the device was time critical. For those devices which are not time critical and might be shared, it was felt that if it was feasible for preemption to be implemented on these devices, UNIX would provide for it. Therefore, since real-time processes were viewed as highest priority by UNIX, they would be given the device. It was further felt that changes in this area if possible were outside the realm of process control and too time consuming for returns gained at present.

VI. IMPLEMENTATION

Implementation was performed in essentially two steps. The first was the implementation of the process controller as designed. This was primarily concerned with interfacing the new process controller with the UNIX operating system as it was received. Essentially testing the feasibility of the design concept was accomplished in the first stage. The next stage of implementation was to interface the new process controller into the new system containing those modifications made to implement multiprocessing and the new memory manager. Both stages required certain modifications or additions to the original design. A summary of these modifications is presented in this Chapter as well as the actual method employed to implemement the design.

A. GENERAL IMPLEMENTATION

For the most part, implementation went smoothly except in the area of scheduling. Once the utilization of the C language was understood, and the exact flow of code was determined, minute one line code changes consisting of unsetting a bit in the new "p_rtflag" entry of the process control block was all that was necessary in the following UNIX routines: EXEC, WAIT, EXIT, and NEWPROC. The routine SCHED required several lines of modification which called for the examination of "p_rtflag" prior to assignment of a process to be swapped out. The implementation of the two system calls, "rtime" and "nonrtime" (see Appendix C),

required the creation of two routines call RTIME and NONRTIME, as presented in Appendix B. Only skeleton versions of these routines were able to be completed in the first stage, the rest being left until stage II. Two short assembly language routines also had to be added to the system library to allow for use of system calls in higher level languages. Several lines of code were introduced into the UNIX routine, SWTCH, to allow for the three possible cases of comparisions between ready processes when searching the process block list. UNIX already provided a method for the case of comparing nonreal-time processes for the highest priority. This was the utilization of a C language "for" loop which started at the process block after the process block of the process last selected, continued through all the blocks and ended with the last block selected. This involved comparing each process priority against the highest priority encountered to that point. If a higher priority was found, a pointer was set to this porcess's PCB, and the highest priority encountered was changed. The only change required was to check for a real-time process, and if either of the two processes being compared at that time had the real-time bit of "p_rtflag" set, then that process was chosen as having the highest priority whether it did or not. If both were real-time or both were nonreal-time then the method implemented by UNIX was allowed to be used without interruption.

B. STAGE I - REVISIONS REQUIRED

1. Introduction of Safety Device

a. Problem

If a real-time process went into an infinite loop while executing in the background mode (independent of terminal control), the process would monopolize the processor and could not be stopped short of halting and rebooting the entire system. The normal method of termination of background processes, utilization of the "kill" command, would have no effect in this case because the process controlling the interaction with the terminal was in a lower priority class than the real-time process. Therefore, this process was never selected for execution due to the ready real-time process always being selected. Thus, implemention of the "kill" command could not be used because it could never be read into the system as long as the real-time process was running. This problem did not occur when a real-time process was running in the foreground mode, due to termination being initiated directly from the terminal through an interrupt from the rub out key.

b. Solution

One solution considered was to make the interactive process (SHELL) real-time. This, however, would mean the SHELL process for each terminal would be locked in memory. In addition, any real-time process would have to compete with each of these processes which would lead to a severe degradation of service for real-time processes. Therefore, this solution was unacceptable.

Another solution and the one adopted was to allow for a controllable pause to occur when a real-time process had run continously for a set amount of time. During this pause, the policy of selection between real-time processes and nonreal-time processes for assignment to the CPU would be reversed. Thus, the highest priority ready

nonreal-time process would be selected for running. Therefore, eventually, the "kill" command would be able to enter into the system to halt the real-time process. Duration of the pause and length of time for continuous real-time execution would both be made adjustable even to the point of having no pause at all. Thus, the controls could be made as stringent or lax as desired, enabling the system to cater to any particular requirements of a real-time process.

c. Changes Made

A counter was established as the timing mechanism to be used. This counter was incremented with every pass through the UNIX routine CLOCK. CLOCK functions in conjunction with the hardware clock interrupt and is entered every one sixtieth of a second. A maximum value was set for the counter, and an automatic reset occurred upon reaching this value. In addition another variable was introduced which was set to the maximum amount of time a real-time process would be allowed continuous CPU utilization.

Alteration of the code contained in the SWTCH routine was also required to implement the reversal of the selection policy at the correct times. This was done by comparing the counter against the variable indicating the maximum amount of time allowed. This was done whenever a comparison between a nonreal-time process and a real-time process was being made. In other cases it was ignored. If the counter had not reached the maximum, then the real-time process was chosen over the nonreal-time, and the opposite occurred when the maximum had been exceeded. In addition, since this restriction was only to be implemented in cases of continuous CPU utilization, code was also added to SWTCH

which provided for the resetting of the counter each time a nonreal-time process was selected to run on the CPU. However, if the counter had exceeded the maximum allowed for real-time execution, it was not reset. In this case resetting would occur in the clock routine when the counter reached its maximum value. The difference between the maximum counter value and the maximum time allotted for continuous real-time execution represented the pause during which nonreal-time processes, if ready, would run.

Thus, initialization of these variables could be used to make the system as responsive to real-time processess as desired. If the automatic reset of the counter was less than the maximum value for real-time execution then, a real-time process would always maintain its status. If the maximum time for real-time was zero or negative, then nonreal-time processes would be selected to run first.

C. STAGE II - REVISIONS REQUIRED

1. Inequality of CPU's

a. Problem

To take advantage of the faster memory cycle time, real-time processes are moved into the private MOS memory assigned each processor by the memory manager. However, this portion of memory is not accessible to the other processor. This restriction is significant in that each processor has a different configuration of peripherals accessible to it. Thus, any process would only be able to utilize devices accessible to the CPU on which it was

running at the time the request was made. For normal processes which were present in the memory shared by both processors, this problem was solved by running the process on the processor which had the device it was requesting. However, with real-time processes in the private memory of each processor, this method could be employed only by moving the process into the shared memory at the time the device was requested. This moving of the process was time consuming and undesireable.

b. Solution

Handling of real-processes would be unchanged, and the responsibility was placed on the user to specify which processor he desired to run on. Once assigned a processor, a real-time process would only run on that processor until termination.

c. Changes Made

Code was added to the RTIME routine to accept the number ID of the CPU to be run on as an argument to the "rtime" system call. Checks were also established to insure valid ID was passed, and this information was forwarded on to the memory manager for the actual moving into the private memory.

VII. CONCLUSIONS AND RECOMMENDATIONS

The process controller, as is, provides for the concurrent execution of real-time, timesharing, and background process. Real-time is given preference and run whenever ready. Timesharing processes and background processes compete with each other for time not utilized by the real-time processes with interactive type processes given higher preference. The process controller is flexible and can easily be adjusted to future requirements. That is, the process controller has the capability of handling more than one real-time process. In addition, the selection scheme can be modified through setting of two variables to make the system as responsive as desired toward real-time processes. The implementation of the processor controller was also able to be completed with a minimum amount of code and made maximum utilization of existing UNIX techniques.

Thus, the process controller, as implemented, performed the desired goal in that it provided for the concurrent execution of real-time, timesharing, and background (batch) processes. All subgoals presented in the design were also able to be upheld, for the most part, in the final product.

However, there are several areas which require follow on work. For one, due to the limited time available, thorough testing was not able to be performed on the new process controller. Therefore, this could be done as well as a study of the actual CPU efficiency. In addition, since system use was still restricted, once the system becomes fully operational and different job mixes begin to occur, a

reevaluation of the present scheduling method might be necessary or desirable to conform the process controller more closely to the job stream. Dynamic alterations in the selection policy might be desirable for certain peak periods of the day. In conclusion, there are the final touches to be added in forming the batch processing mode, that is, the card reader/printer type input/output and the initialization of the default priorities for the different class or classes of batch jobs. The mechanisms for doing so are available within the present operational structure of UNIX and only require implementation.

APPENDIX A. PROCESS CONTROL BLOCK

A1. UNIX

A C language structure called "proc" is used as the process control block under UNIX. This structure contains fourteen individual entries which reflect the status of the corresponding processes at any given time. Since this structure was directly involved with scheduling processes, it was the data structure of most importance in implementing the new design. The function of each of the entries in this structure is as follows:

p_stat is used to indicate the state of the process with regards to exectuion. Under UNIX a process could be in one of five states:

* SSLEEP means the process has been put in a wait state with a high priority and can not be run.

* SWAIT means the process has been put in a wait state with a priority greater than or equal to zero and can not be run. This state is utilized extensively when a process is waiting for completion of I/O.

* SRUN means that the process is ready to be executed.

* SIDL means that the process is active but does not

belong in any other status.

* SZOMB means that this process has terminated, but the information in the PCB is still required for other uses.

p_flag is used to maintain the status of a process with regards to memory. It can take on one of four values:

* SSYS means the process is a special system process.

* SLOAD means the process is loaded in main memory.

* SLOCK means that the process is required to remain in memory and should not be swapped out.

* SSWAP means the process is resident in the swap file.

p_pri is used to maintain the priority assigned to this process.

p_sig is used to indicate whether certain signals are to be ignored or held. Odd values will cause them to be ignored and even will cause them to be caught.

p_uid is used to indicate the owner of this process. This unique ID is given to the user at the time of login.

p_time is used to record the amount of time the process has been in main memory or in the swap file and is zeroed when the process transits from one to the other.

p_ttyp is used to indicate the terminal to which the process belongs.

p_pid is used to hold the unique number ID given to this process at creation.

p_ppid is used to hold the p_pid of this process's parent.

p_addr is used to store the address of the process in memory or in the swap file depending on where the process is. It is not the address of the first portion of executable code but is the beginning of the u-vector block, in particular the "user" structure. This entry maintains the link between the PCB and the process image.

p_size is the size of storage the process image requires.

p_wchan contains a number value which represents the reason that this process was placed in a wait state (SSLEEP or SWAIT). It can represent the number of a device, process, or any other feature which may have caused this transition.

p_textp is a pointer which indirectly links the process to any other process's code which it might be sharing.

A2. REVISIONS MADE

The only differences between the process control block used by UNIX and the one used by the new process controller was the addition of a new integer entry called "p_rtflag".

p_rtflag is used to indicate whether or not a process is real-time. It contains a value of zero if the process is not real-time, and a value of one if it is.

APPENDIX B. SYSTEM ROUTINES

This Appendix presents those routines within the body of the UNIX operating system which directly comprised the process controller. Only those felt to be most important to the understanding of changes made are listed. In addition, the routines required to be added to form the new process controller are also presented.

B1. FORK

FORK is a UNIX routine related to the "fork" system call available to the user. FORK searches the list of process control blocks for an empty block. If one is found, the routine NEWPROC is called to create a duplicate process. NEWPROC returns a value of zero for one process, and FORK chooses that one to be the parent. The ID number of the child process is loaded into register zero, and FORK returns from the system call. A value of one is returned for the other process, and FORK chooses that one as the child. FORK zeros all the time recorders in this process's u-vector, zeros R0, and returns from the system call to the child.

B2. NEWPROC

NEWPROC is a UNIX routine which actually creates a duplicate copy of a process. It is called mainly from FORK, but is also utilized by MAIN when the system is being

bootstrapped into operation. The first thing NEWPROC does, is search the PCB list for the first available empty block. This is a duplication of effort when the call comes from FORK, but needs to be done because no parameters are passed between the two routines. Next, NEWPROC saves a pointer to the old process which was running and starts entering values into the new process block: sets "p_stat" to run; sets the loaded bit in "p_flag"; makes duplicate entries for user ID, terminal, and shareable sections; sets ID to next number in sequence; sets the value of the "p_ppid" to the ID of interrupted process; and zeros the "p_time". Following this, it increments necessary open file counters, shareable file counters, and directory user counters to reflect the use by the new process. NEWPROC then calls another routine to save the user state in the u-vector. It sets a pointer to the u-vector in the process block of the new process; saves the address of the old process; calls the memory manager routine to request allocation of an amount of memory equal to the size of the old process. If memory is available, it sets "p_addr" of the new process block to the address returned and calls a routine to copy the old process code, etc. into the new area. Finally, NEWPROC resets the pointers for the old process and returns. If space is not available in core; that is, the memory manager routine called returns a value of zero, then a copy of the process is swapped out to the disk and appropiate entries are made in the PCB, and then NEWPROC returns.

B3. SCHED

SCHED is a UNIX routine associated with the process control block zero and is used to enforce the high level scheduling policy. SCHED searches the process block list checking the "p_time" values of all ready processes

("p_stat" equal SRUN) which are not loaded into core. It selects the process which has the greatest "p_time" value. This is the process which was on the swap file for the longest period of time. If there are no processes on the swap file, SCHED sets a variable called "runout", and goes to sleep on this variable using PCB 0 with a very high priority. Then when WAKEUP is called, it checks this variable and the process list to determine if anything is now out on the swap file, and if there is, it does a wake up on "runout" which causes SCHED (proc 0) to be awakened. Due to its high priority, proc 0 is selected and run by SWTCH causing SCHED to go through its loop again. If there is a process waiting to come in, SCHED checks its process size, increases the size to reflect the size of any shareable sections not yet in core, and calls the memory manager to request memory space. If available, SCHED calls a routine to swap the shareable section into the core. It then adjusts pointers in the process block to maintain the link, and calls this routine again to swap in the process. SCHED then calls MFREE to deallocate the swap file space; sets "p_time" to zero; sets the SLOAD bit in "p_flag"; and sets "p_addr" to the address of the process in core. Then SCHED loops back to the beginning of itself and executes the same code again. It continues this looping until there are no more processes on the swap file, none have been on there longer than three seconds, or no more core is available and the longest process in core has been there for less than two seconds.

If no core is available when a process is ready to come in, SCHED checks for processes which are in core, not locked, not a system process, and in a wait status. If one is found it is swapped out, and SCHED loops back again to the beginning of its code, finds the process waiting the longest on the swap file, and checks the memory which now reflects the additional space just vacated by the process

which was swapped out. If enough core is still not available then the process is repeated.

If there are no more processes in a wait state in memory, and a process has been waiting for greater than three seconds on the swap file, SCHED becomes less restrictive in its search. It now searches through the process list looking for the process which has been in memory the longest; is not locked in memory; is not a system process; and is in a run or sleep state (waiting on a system function). If a process is found and has been in memory for greater than two seconds, it is swapped out. Then as before SCHED cycles back to the beginning of its code. If no processes meeting these requirements are found or the longest time in memory for all processes is less than two seconds, then SCHED goes to sleep on a variable called "runin". The functioning of "runin" is similar to the functioning of "runout" except that it is checked in the SLEEP routine whenever a process is placed in the wait state instead of in WAKEUP.

B4. SWTCH

SWTCH is the UNIX routine which implements the algorithm to select the next process to run on the CPU. It begins by saving the state of the user which just used the CPU and by changing the u-vector utilized to the one belonging to process 0. This is done to insure the system has a valid u-vector area for use. Previous to this, the system had been using the u-vector belonging to the process which was running. However, due to the process being forced to relinquish the processor for some reason and to the asynchronous operation of the system, the system can not be certain that this process's u-vector will be available for

use during the system's search for a new process to run. This is especially true, if the process has terminated or is available for swapping out. Therefore, as an insurance against the possibility of not having a valid u-vector area for use for interrupts, return from interrupts, etc, the system reverts to its own u-vector area for use during this transition period between processes.

Having completed this function, SWTCH begins a search of all fifty process control blocks. It starts with the block immediately after the last one picked by SWTCH, and compares the priorities of the processes which are ready and loaded into core ("p_stat" equals SRUN and "p_flag" loaded bit set). If no process is found, the CPU is set to an idle state. An interrupt will cause the CPU to return from the idle state. When this happens, SWTCH loops back to the beginning of its code and performs the search again starting with the exact same PCB which it used before setting the CPU idle. When a process which meets the required criteria stated above is found, SWTCH, with the help of RETU, sets the execution envirnoment for this process in the system. Upon completion of this task, it returns a value of one to its caller.

B5. SLEEP

This routine is called with two integer value arguments; one identifies the reason for the call, and the other represents the priority to be attached to it. After saving the Processor Status Word (PS), SLEEP checks the priority value. If it is less than zero, SLEEP sets the "p_wchan" of the last user process which was running ("u.u_procp->p_wchan") equal to the reason passed, its priority equal to the one passed, and its "p_stat" equal to

SSLEEP. It then calls SWTCH to select a new process for running and on return from SWTCH, returns to its caller.

If the priority value passed is greater than or equal to zero, SLEEP takes a little different course of action. It first calls ISSIG to determine if this process has specified an address at which an interrupt is to be simulated. If this is the case, it returns this non zero value to SLEEP which sets up this section of the process to receive control and returns (see system call "signal" Ref. 17). If a value of zero is returned from ISSIG, then SLEEP sets "p_wchan" and the priority as before but sets "p_stat" to SWAIT. The value of "runin" is checked, and if set, a call to WAKEUP is made with "runin" as the argument. SWTCH is then called, and the return is made.

B6. WAKEUP

This UNIX routine is called with an integer value argument which represents a device, channel, or some other reason for which a process might have been put to sleep or otherwise placed in a nonready condition. A call to WAKEUP indicates the argument passed is now available for use. WAKEUP searches the process block list checking the "p_wchan" position for a match in value with the argument passed. When a match is found, the process's "p_stat" is set equal to SRUN. This is done for all processes waiting on this argument. Having completed this, WAKEUP checks the "runout" variable and if set, does a wakeup on everyone waiting on the "runout" channel. Normally only process 0 which corresponds to the UNIX routine SCHED is waiting on this channel. WAKEUP does a normal return after this last chore.

B7. EXEC

EXEC is the UNIX routine which corresponds to the system call "exec". Upon execution, this routine checks the file listed as an argument to insure it is eligible for execution. This involves searching for the file to determine if it exists and checking to insure it is an object code file and execute permission is granted. It next, checks out the arguments required by the file to insure correctness. Then if everything checks out, EXEC, reads in the text, data, and stack sizes and compares them against the sizes presently being utilized by the calling process. This is to insure enough core is available for the new process, and any excess core not needed is released. After the adjustments have been made to match up the different areas as far as size is concerned, the text, data, and stack segments are copied into core, and the memory registers are set. Then all appropiate entries are made in the user vector and process control block to reflect any required changes in the form of the process. At completion of this, the routine clears the registers for this process and returns having already set the new return address and cleared the program counter.

B8. EXIT

EXIT is the UNIX routine which corresponds to the system call "exit" and is automatically envoked upon termination of a process. This routine performs the housekeeping functions necessary after a process terminates. First, it unmasks all signals which may have been masked by the process and allows them to clear. It next closes all open files and frees the portions which were being shared. It then secures swap

space and swaps out the user vector block placing the address of this vital information in the "p_addr" area of this process's PCB. The routine then frees all memory belonging to the process and sets the PCB status to indicate a special dormant state (SZOMB). It next searches the process list looking for the process's parent. If found, a wakeup is performed on the parent and process one (INIT). If a parent is not found, then EXIT performs a wakeup only on INIT. It next searches the PCB list again looking for processes to which this process was a parent. If any are found, their parent ID values are changed to that of process one, INIT. At the completion of this, EXIT turns control over to SWTCH to select a new process for running.

B9. WAIT

WAIT is the UNIX routine corresponding to the system call "wait". Upon entry, this routine searches the PCB list checking the parent ID's looking for a match with the current user process running on the processor. If a match is not found, an error is given. If found, a check is made to determine if the process found is in a dormant state. If not, then the routine sets a variable to indicate that a child has been found but was not in the dormant state. The search then continues for possibily another process meeting the criteria. If the process found was in the dormant state, then the WAIT routine copies the child's u-vector from the swap device into memory. It then transfers all the timing information and the value of the child's R0 into the parent's u-vector. It then zeros out the child's PCB, makes the PCB available for reassignment, and returns.

If a child was found but was not in the dormant state, the WAIT routine calls SLEEP with the parent's PCB address

as the argument. This in effect puts the parent to sleep on its own address with the return address from the sleep state being set to a position within the WAIT routine. Thus, upon being awakened, the parent will be assigned the CPU, and its program counter will be pointing to the WAIT routine. This will cause the wait routine to resume processing after the SLEEP call, at which point the WAIT routine loops back to beginning of its code and performs the search again.

B10. CLOCK

CLOCK is the UNIX routine which interfaces the operating system with the hardware clock of the processor. At every clock pulse interrupt, this routine is called to perform the various timing functions required. This includes incrementing the "p_time" of all the process control blocks, the usage time in the u-vector of the running process, the priority of the running process, and doing other housekeeping functions in regards to time. It also performs a wakeup on "runin" and on any other process that had performed a "wait" based on time.

B11. TRAP

TRAP is a C language routine which handles the software portion of routing interrupts to the proper routines. It handles interrupts caused by the following: a bus error; an illegal instruction; a bpt-trace trap; iot trap; emulator trap; a system function call; a programmed interrupt; a floating point exception; memory management violations; and memory management traps. The reason for the interrupt is passed as an argument, and after saving the floating registers of the user, TRAP checks this value in combination

with other factors to determine the proper procedure to execute. After the corresponding routine has returned, TRAP performs the task of resetting the present user's priority back to the default of 100 or greater if user has so specified. If this process has had more than fifteen interrupts handled by TRAP, the process's priority is incremented and a return is made to SWTCH instead of this process.

B12. SAVU

SAVU is an assembly language routine whose primary function is to save the stack pointer and the value of register five. These are placed in the "u_rsav" array of the user vector. The importance of register five is that it points to the place on the stack where the current function's arguments are stored. The stack pointer provides access into the Kernel stack residing within the u-vector.

B13. RETU

RETU is an assembly language routine whose primary function is to restore the execution environment for a process. It starts by setting the system pointer, Kernel Data Space Address Register 6, to the value of the process's "p_addr". KDSA6 is the register used by the system to indicate which u-vector is to be used by the processor while executing a process. After setting this pointer, RETU performs the reverse of SAVU and restores the stack pointer and value of register five.

B14. SYSENT

SYSENT is a UNIX routine which is used to convert system calls into calls on the actual UNIX routines which perform the functions. It is an integer function which returns the routine's address to the calling function.

B15. RTIME

RTIME is a C language routine written to implement the new process controller. It corresponds to the "rtime" system call utilized to request real-time status. Upon execution, this routine first checks the validity of the processor ID passed in R0. If invalid, the routine returns an error code of minus thirty three (EUNKCPU) through the "u_error" entry in the "user" structure. If it is valid (0 or 1), this routine records this value for future use. The routine next searches the entire process block list counting the number of real-time processes currently running on the processor requested. If this value is greater or equal to the maximum allowed (NRTIME), RTIME returns an error value of thirty two (RTBUSY) to indicate the busy status. If this value is not greater than the maximum, the RTIME routine sets the real-time bit and corresponding processor requested bit in the process's PCB. It then calls the memory manager to move this process into the processor's private memory. If no errors occur, RTIME returns a value of zero to the calling process. At present, these are the only checks made in RTIME. However, this routine is foreseen to absorb other checks and functions related to real-time as the new system matures.

B16. NONRTIME

NONRTIME is a C language routine written to implement the new process controller. It corresponds to the "nonrtime" system call, and its sole purpose is to unset the real-time bit in the calling process's PCB. This routine's functioning is also foreseen to grow and change as the system matures.

APPENDIX C. SYSTEM CALLS

UNIX provides system calls which allow user programs to communicate with the operating system. These calls provide the user with a variety of functions which would otherwise be unavailable. System calls are utilized within a program as are any other function calls. A complete listing of these calls can be found in Ref. 17. A brief summary of the more important ones pertaining to process control are presented here. The two system calls developed in conjunction with creating real-time processes, "rtime" and "nonrtime", are also presented.

C1. "fork"

This system call enables a user to create a new independent process during the execution of his program. This call returns a value of zero to the child process and a nonzero value to the parent process ("p_pid" of the child). If the new process was unable to be created, a value of minus one is returned. This system call is usually used in conjunction with the system calls, "exec" and/or "wait".

C2. "exec"

This system call enables a user to replace the program making the call with another. This other process is created from the file whose name was passed as an argument.

Execution of this new process will start at the beginning of its code. The new process inherits all open files and any special provisions made for handling certain interrupts. However, the original image is destroyed upon execution of this command. Access privileges for this new process are the same as the owner of the file executed and are not those of the user.

C3. "wait"

This system call enables a user to delay execution of a process until its child terminates. Return from the call is immediate if a child has terminated since execution of the last "wait" call or no child exists. In the latter case, the error bit is also set. If the desire is to wait on the termination of several children, a "wait" call must be performed for each.

C4. "exit"

This system call is used to terminate a process and does not return to the calling process. This call causes all the process's files to be closed and notifies the parent if it executed a "wait" call.

C5. "kill"

This system call sends a signal to the process specified by the process number passed in register zero. The sending and receiving processes must have the same controlling terminal. Only the super user (user having unrestricted

privileges) is permitted to utilize this command for a process with a different terminal. This system call is used by the SHELL to implement the kill command when halting a process running in the background mode.

C6. "rtime"

This system call enables a process to request real-time status. Granting of this status locks the process into private high speed memory, grants immediate access to the processor, and grants head of the line privileges for all I/O devices. However, the process is restricted to utilization of only one processor and associated peripherals. The argument passed at the time of the call specifies which processor is desired (a value of 0 for processor A and a value of 1 for processor B). Any value other than a zero or one used as an argument will cause denial of the request. The error bit (c-bit) will be set in this case. It will also be set if the processor requested is presently busy with the maximum allowed number of real-time processes. In C language, a value of zero is returned if request granted; a value of thirty two is returned if processor busy; and a value of minus thirty three is returned to indicate invalid processor requested. Real-time status is automatically cancelled at process termination or through use of the system calls "exec" or "nonrtime". Use of the "fork" system call retains the real-time status for the parent only, but the child must reinitiate the "rtime" call if real-time status is desired.

C7. "nonrtime"

This system call enables a process to cancel its real-time status if desired. There are no error values returned from this call.

LIST OF REFERENCES

1. Ritchie, D. E. and Thompson, K., "The UNIX Timesharing System", Communications of the ACM, v.17, no. 7, p. 365-375, July, 1974.

2. Hawley, J. A. and Meyer, W. B., MUNIX, A Multiprocessing Version of UNIX, M. S. Thesis, Computer Science Group, Naval Postgraduate School, Monterey, California, 1975.

3. Kruse, D. M. and Winther, J. C., Virtualization of The PDP-11/50, M. S. Thesis, Computer Science Group, Naval Postgraduate School, Monterey, California, 1975.

4. Marsh, M. T., Memory Management For Paged, Hierarchical Memory in A Multiprocessing Computer System, M. S. Thesis, Computer Science Group, Naval Postgraduate School, Monterey, California, 1975.

5. Madnick, S. E. and Donovan, J. J., Operating Systems, McGraw-Hill Book Company, 1974.

6. Katzan, H., Jr., Operating Systems, Van Nostrand Reinhold Company, 1973.

7. Comptre Corporation, Operating Systems Survey, Auerbauch, 1971.

8. Association for Computing Machinery, Comparative Operating Systems, 2nd ed., Prandon/Systems Press, Inc., 1970.

9. Varney, R. C. and Gotterer, M. H., "The Structural Foundation for an Operating System", Computer Journal,

v. 16, no. 4, p. 357-359, November, 1973.

10. Organick, E. I., The Multics System, M.I.T. Press, 1972.

11. Digital Equipment Corporation, PDP 11/45 Processor Handbook, 1974.

12. Digital Equipment Corporation, PDP 11 Peripherals Handbook, 1974.

13. Bell Laboratories, C-Reference Manual, 1972.

14. Bell Laboratories, UNIX Assembly Language Manual-PDP 11, 1972.

15. Kernighan, B. W., Programming in C - A Tutorial, Bell Laboratories, 1974.

16. Kernighan, B. W., A Tutorial Introduction to The ED Text Editor, Bell Laboratories, 1974.

17. Thompson, K. and Ritchie, D. M., UNIX Programmer's Manual, 5th ed., Bell Laboratories, 1974.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 72 Computer Science Group Naval Postgraduate School Monterey, California 93940	1
4.	LT Belton E. Allen, USNR, Code 72An Computer Science Group Naval Postgraduate School Monterey, California 93940	1
5.	LT Theodore C. Kral, USN Kieners Lane Pittsburgh, Pennsylvania 15205	1
6.	LTJG Gary M. Raetz, USN, Code 72Rr Computer Science Group Naval Postgraduate School Monterey, California 93940	1